SproutCore Themes Specification

Version 1.0
November 18, 2010

Authors:

Yehuda Katz wycats@sproutcore.com

Requirements

Features

CSS Theming

In many cases, a view can have a single HTML representation, and have its look and feel customized using CSS. In order to enable this case, SproutCore Theme objects have a unique HTML class name, which is included in the root element for views using that theme. The theme's CSS rules can use those classes to create unique names.

HTML Customization

In some cases, themes may want to control the HTML representation of a particular view. SproutCore Theme objects can customize the HTML representation of views it wants to customize through Render Delegates. For more information, see the Render Delegates specification.

Sub-Themes

SproutCore views can specify their themes using the theme's identifier. Normally, the theme is looked up in the global registry of themes. For instance, if the theme `'ace'` is specified, SproutCore will find the theme named `'ace'`. In some cases, it can be convenient to use a theme name that resolves to a different Theme objects depending on the theme of the parent view. For instance, you might want to specify that an overlay uses the `'popover'` theme, which would be different if its parent view was using `'ace-desktop'` or `'ace-tablet'`.

Constraints

Reasonable Defaults

In general, views should not have to specify their theme. Instead, they should inherit their themes from their parent view. In addition, it should be possible to set a default view for an application, which top-level views should use if no theme is specified.

Global Registry

It should be extremely simple to create a new theme, specify its class name, and register it with SproutCore. Since many views can be themed with CSS only, it should be very easy to create and register a simple theme that just provides a class name to add to views using it.

Changing Themes at Runtime

It can sometimes be useful to change the theme of a particular view at runtime. For instance, a view might visually change in landscape mode, but still have exactly the same display properties. In this case, it should be possible to modify the theme's name at runtime, and have it replace the HTML representation of the view with its updated representation.

Inherited HTML Representations

When a Theme class is extended, the new theme inherits the renderDelegates associated with the original theme. This means that the HTML representations defined for the parent's theme will still apply to views using the extended theme unless explicitly overridden.

Inherited Class Names

When a Theme class is extended, the new theme's class names include both the parent's class names and its own. This means that any CSS defined for the parent's theme will still apply to views using the extended theme unless explicitly overridden.

Specification

Theme Lifecycle

This section describes how themes are created and used by views.

Base Theme

SproutCore's built-in views define their default HTML representations as renderDelegates on `SC.BaseTheme`. As a result, when creating a theme to customize built-in views, theme developers should generally extend `SC.BaseTheme`, so they only have to modify the HTML representations of views that would be different in the custom theme from the default representation.

Creating a Theme

A theme is first created and registered by extending `SC.BaseTheme`

```
MyApp.MyTheme = SC.BaseTheme.extend({
  name: “my-theme”,
});
```

This creates a new theme called `‘my-theme’`, extended from `SC.BaseTheme`. This means that it will inherit `SC.BaseTheme`'s CSS class (`‘sc-base’`) and its Render Delegates.

Creating a Subtheme

Creating a subtheme is almost identical to creating a theme, except that the `‘subtheme’` method, rather than the `‘extend’` method is used.

```
MyApp.MyTheme.Dark = MyApp.MyTheme.subtheme({
  name: “dark”
});
```

This creates a new theme, extended from `MyApp.MyTheme`, which is also added as a subtheme to `MyApp.MyTheme`. You would use the `MyApp.MyTheme.Dark` subtheme in a child view of a view themed with `MyApp.MyTheme`:

```
MyApp.MainPage = SC.Page.design({
  mainPane: SC.MainPane.design({
    childViews: [‘image’],
    themeName: ‘my-theme’,

    image: SC.ImageView.design({
      themeName: ‘dark’
    })
  })
});
```

In this case, the `ImageView` will use the `MyApp.MyTheme.Dark` theme, which it finds by calling `MyApp.MyTheme.find(‘dark’)`, which finds the theme we registered above.

The ImageView will get `‘sc-base’`, `‘my-theme’` and `‘dark’` as CSS classes. It will also have the `‘sc-image-view’` class, which it gets by extending `SC.ImageView` (`‘sc-image-view’` is declared as `classNames` on `SC.ImageView`).

Default Theme

Applications can specify the default theme for root views without an explicitly specified theme name by using `SC.defaultTheme`. Child views without an explicitly specified theme name will inherit the theme from the root views.

```
SC.defaultTheme = ‘sc-ace’
```

renderDelegates

Views can specify that they want to delegate creating and updating their HTML representation by specifying a String `renderDelegateName` property. When rendering, the view will ask its current theme for a Render Delegate with that name, and use it to render and update the view (for more information, check out the Render Delegate specification).

Themes can provide a `renderDelegate` for the views they want to theme. Themes also inherit `renderDelegates` from themes that they extend. An example of creating a new set of custom views designed to be themed:

```
// Create a new base theme for all custom views in the
// SCUI framework
SCUI.BaseTheme = SC.BaseTheme.extend({
  name: “scui”,
});

SC.Theme.addTheme(SCUI.BaseTheme);

// Create a ‘dark’ subtheme of ‘scui-base’
SCUI.DarkTheme = SCUI.BaseTheme.extend({ name: ‘scui-dark‘ });
SCUI.BaseTheme.addTheme(SCUI.DarkTheme);
SCUI.BaseTheme.registerRenderDelegate({
  render: function(dataSource, context) { /* render */ },
  update: function(dataSource, jquery)  { /* update */ }
}, ‘scui-button’);

// the dark theme inherits the ‘button’ renderDelegate
// from its parent theme
SCUI.DarkTheme.registerRenderDelegate({
  render: function(dataSource, context) { /* render */ },
}, ‘scui-image’);

SCUI.ButtonView = SC.View.extend({
  renderDelegateName: ‘scui-button’
});

SCUI.ImageView = SC.View.extend({
  renderDelegateName: ‘scui-image’
});
```

Here, the custom views are delegating the rendering of the HTML representation to the theme. In my app, I could use these views.

```
MyApp.MainPage = SC.Page.design({
  mainPane: SC.MainPane.design({
    childViews: [‘image’],
    themeName: ‘scui’,

    image: SCUI.ImageView.design({
      themeName: ‘dark’
    })
  })
});
```

All the user of the framework needs to do is specify the theme they wish to use, and the appropriate renderDelegate for that theme will be chosen. If someone wanted to create a custom theme for the SCUI set of views, they would just need to register a renderDelegate with the names `‘scui-image’` and `‘scui-button’`, and when the developer chose their theme, SproutCore would automatically use those delegates.

This can also be used to add support for third-party custom views in built-in themes like Ace.

```
SC.Ace.registerRenderDelegate({
  render: function(dataSource, context) { /* render */ },
  update: function(dataSource, jquery)  { /* update */ }
}, ‘scui-image’);
```

This satisfactorily decouples the custom view itself from its rendering, making it easy for those developing themes to add support for views, whatever their source, and making it easy for those developing custom views to add support for popular themes.